Everything you always wanted to know about HPFS...

...but were afraid to ask. By Max and Sandy Eidswick

You've probably seen many articles about the High Performance File System (HPFS) and plenty of coverage in OS/2-related books. So why yet another one? Because, more often than not, the articles written about HPFS delve so deeply into the file system's data structures, Fnodes, and bitmaps, that HPFS's basic benefits are overlooked—a case of not being able to see the forest for the trees.

Let's look at the forest by discussing a few of the most common questions about HPFS.

Where did HPFS originate?

Back in 1985, IBM and Microsoft agreed to jointly define and ship Operating System 2, the next generation of DOS. OS/2 development continued during the mid-to-late 1980s as a cooperative effort involving design engineers, developers, and programmers from both companies.

Part of this cooperative development was the design of a new, advanced file system. Network servers needed a file system better suited to efficiently managing huge amounts of data on large disks than DOS's File Allocation Table (FAT) system. They needed a file system that was both reliable and capable of supporting the multitasking server environment.

HPFS provided the solution. It was designed by Gordon Letwin of Microsoft's network division as an advanced file system for Microsoft's LAN Server product. (This original version of HPFS is what we know today as HPFS386.)

When OS/2 1.2 shipped, it included its own desktop version of HPFS, providing the desktop with a file system that could manage large volumes of disk space more efficiently than the FAT system and robust enough to support OS/2's multitasking ability on the standalone PC.

The original HPFS386 was written in assembler language. The newer HPFS, shipped with all versions of OS/2 since 1.2, is written instead in C, most likely because C was more comfortable for those maintaining code than was assembler language.

How does HPFS differ from FAT?

The key difference between HPFS and FAT is the way each goes about storing data on a disk. To understand the difference, let's first explore how storage space is allocated on a disk formatted under FAT.

FAT

FAT is short for *file allocation table*. The file allocation table is just that: a table containing entries that point to the physical location on the disk of the cluster or clusters in which all or part of a file is stored. A cluster is a grouping of 512-byte disk sectors (typically the smallest element of disk storage addressed by PC-based device drivers). Files are simply chains (or linked lists) of clusters in the file allocation table. The clusters containing the sectors of any given file are, more often than not, scattered over the disk.

What is most important to understand about the file allocation table is that its size is limited to no more than 64K entries. Therefore, as disks have gotten larger, the size of the minimum cluster has had to increase, too, in order to keep the number of table entries below the 64K threshold. Table 1 shows the relationship between the size of a disk or volume and the minimum cluster size permissible using the file allocation table.

Every file on your disk consists of at least one cluster. As the size of a file exceeds one cluster, a new cluster is allocated and added to the end of the chain. For each file, an average of half the last cluster will be unused. This unused space is known as slack or lost space, and as your disk size grows, the proportion of it that is consumed by slack space grows dramatically, because of the ever-increasing size of the minimum cluster.

To examine the relationship between disk or volume size

Table 1: File allocation table relationship between disk or volume size and minimum cluster size.

Disk or Volume Size	Minimum Cluster Size
32 MB	½ KB (512 bytes)
64 MB	1 KB
128 MB	2 KB
256 MB	4 KB
512 MB	8 KB
1 GB	16 KB
2 GB	32 KB
4 GB	64 KB

and lost space, we looked at several of the machines in our office in terms of the drive's size and the number of files it contained. We use them here as an example. While a mix of files in other settings will vary from machine to machine, the general concept we present here with regard to lost space seems to hold true across the board. That is: As the size of the volume doubles so does the percentage of total disk space lost.

Table 2: Slack space in various sized disk volumes under the FAT system.

Volume Size	Cluster Size	Avg. No. of Files	Estimated Slack Space	Slack Space as % of Total Space
32 MB	½ KB	1,250	314 KB	1 %
64 MB	1 KB	2,500	1.25 MB	2 %
128 MB	2 KB	5,000	5 MB	4 %
256 MB	4 KB	10,000	20 MB	8 %
512 MB	8 KB	20,000	80 MB	16 %
1 GB	16 KB	40,000	320 MB	32 %
2 GB	32 KB	80,000	1.28 GB	64 %

Table 2 shows the slack space wasted as disk volumes grow larger. It's based on the assumption that a typical 512MB volume will contain about 20,000 files. A volume half that size (256MB) should hold about 10,000 files; one double in size (1GB) should hold twice as many. See how the slack space dramatically increases as drives increase in size—your new 1GB disk drive is not able to hold twice the number of files that your old 512MB disk did. Why don't you get twice the storage? Because of the inefficiencies of the FAT file system.

The FAT file system provided an excellent solution for the limited needs of the early DOS and Windows applications. After all, in the early 1980s, the primary storage media was limited in size itself. First, there was the 180KB floppy, then came the 360KB floppy. Finally, we saw the 80286-class machines with 1.2MB floppies and huge 10MB hard-disk drives. Today, most applications will take at least 20MB of disk storage each, and the operating system may take more than 40MB for system files.

As PC technologies have progressed to provide faster processors, bigger applications, and larger hard disks, the limitations and inefficiencies of FAT-based file access begin to impinge on both the user and the operating system. Today, it is common to find ordinary desktop PCs purchased with 1GB to 2GB hard disks.

HPFS

Armed with an understanding of the mechanics of the file allocation table, it's time to look at how HPFS differs in its approach to storing data on your disk. The key is that the basic storage allocation unit is no longer a cluster. Instead, HPFS stores data using 512-byte sector boundaries—regardless of the disk's size.

To realize the impact this has on a disk, you need only compare two like-size volumes, one FAT and the other HPFS. For example, on a 128MB FAT-formatted volume with a 2K cluster, you need 2,048 bytes (one cluster) to store a file that is only 1,024 bytes long. When the same volume is formatted using HPFS, you need two 512-byte sectors or a total of only 1,024 bytes to store the same file.

HPFS Versions

The HPFS version describes the level of HPFS used to format the volume and has no correlation to the OS/2 version. The versions in use currently are 2.2 for small HPFS volumes, 2.3 for large HPFS volumes, and 2.4 for HPFS386 volumes.

In general, the *hpfs.ifs* and corresponding *uhpfs.dll* (in which the HPFS utilities **FORMAT** and **CHKDSK** are kept) are not dependent upon the operating system release. Normally, however, you want to run the versions that came with the retail version of OS/2 running on your system.

From time to time (between retail versions of OS/2), IBM releases FixPaks that correct a variety of very specific user-reported problems. They are quick fixes that not everyone needs or should apply. While the FixPaks should not break anything, it's always possible. Therefore, we recommend that you keep backup copies of your original *hpfs.ifs* and *uhpfs.dll*, so that you will have them should you need them.

Table 3 takes this example one step further by looking at what the impact will be of storing this same 1KB file on larger volumes with larger cluster sizes. Imagine the toll that downloading thousands of small message files (typically 256 to 1,024 bytes in size) has on a 1GB FAT-formatted volume, where 15Kbytes of space are lost for every 1Kbyte used.

How does HPFS resist fragmentation?

HPFS resists the tendency toward disk fragmentation, which is a part and parcel of a FAT-formatted drive, by virtue of the way that data storage is allocated on an HPFS volume. First, while the smallest storage allocation unit on a FAT-formatted drive is a 512-byte cluster, this is only the case when the volume is less than 32MB in size. By contrast, the only storage allocation unit on an HPFS-formatted drive is a 512-byte sector, regardless of the size of the drive.

Next, HPFS divides each volume into bands of 8MB—again, regardless of the size of the volume. Unlike the file allocation table with its fixed number of entries, HPFS creates as many 8MB bands as necessary to fill up the volume. Each band has its own free-space bitmap, with HPFS keeping statistics on each band. These statistics are used to select the most favorable bands for allocating space to a specific file.

Finally, HPFS attempts to keep a file's allocation localized and contiguous. Whenever possible, the file system preallocates additional contiguous space for a file to grow into, rather than having to store the file in a new location. Fragmentation is minimized by allocating files from separate data bands and preallocating contiguous file space.

Table 3: Comparing slack space for 1KB file stores under both FAT and HPFS.

Volume Size	Minimum FAT Cluster Size	Actual File Size	Lost Space on a FAT Volume	Lost Space on HPFS Volume
128 MB	2 K	1 K	1 K	None
256 MB	4 K	1 K	3 K	None
512 MB	8 K	1 K	7 K	None
1 GB	16 K	1 K	15 K	None

Why can't I access a drive that hasn't been correctly stopped?

Each time you correctly shut down your system, the file system is shut down, its buffers are flushed, and the "dirty read" flag on each HPFS partition is reset to "clean."

An HPFS drive that hasn't been shut down correctly can't be accessed because the file system sees the "dirty read" flag. The flag signals the file system that the drive was stopped before its free-space bitmaps could be updated to reflect changes to the drive's contents. To prevent damage caused by using those potentially inaccurate bitmaps, the file system requires that the drive be checked (using CHKDSK, the HPFS disk-checking utility) before any of the data on the drive is accessed. CHKDSK corrects any flaws and then releases the lock.

Sometimes you may need to run CHKDSK more than once. That's because on an HPFS drive, the primary and secondary disk structures are checked and "cleaned up" in multiple passes of the CHKDSK program. Therefore, if there are errors on a drive, a single pass may not be enough. The program should be run again and again until no errors (or corrections) are reported.

Why aren't there any HPFS floppies?

We can fall back on definitions, according to IBM's Doug Azzarito. Because the HPFS specification says in writing that HPFS does not support removable media, you can't have HPFS-formatted floppies! It's not a case of its being technically impossible (after all, you could trick OS/2 into thinking that a floppy disk is not a removable disk)—it just doesn't make sense.

The more logical reason that no HPFS-formatted floppies exist is because of the file system's use of the lazy-write cache. In order to improve system performance, data is held in memory cache buffers and written to disk when the disk is idle. If the target media is removed before the file system writes the cached data to disk, the data is lost.

However, the data held in the cache buffers is not lost if, prior to removing the media, HPFS is notified. The media itself must do this by providing a signal for HPFS that causes the file system to issue a command to "redetermine" the media. This command tells HPFS to read the floppy's boot sector before each write. Unfortunately, the constant monitoring of the floppy's boot sector makes using an HPFS-formatted floppy more painful than beneficial.

Newer, more sophisticated removable media is capable of notifying the file system of changes in media and avoiding the necessity for the file system to constantly monitor the media's boot sector; someday, they may be HPFS formatted.

What is HPFS386?

Historically speaking, HPFS386 is the original High Performance File System, designed by Gordon Letwin of Microsoft. It was intended to be used exclusively with the Microsoft LAN Server product and was written in assembler language.

HPFS386 uses special, undocumented interfaces to and from the LAN Server code. These speed up disk I/O (input/output) by allowing I/O requests to go directly from the file system to the network adapter. The result is disk I/O with less overhead and speedier response to network client requests for disk access.

Today, HPFS386 is available with IBM's Advanced LAN Server product. While it is compatible with HPFS, it also includes the following features specifically designed for LAN use:

- Access Control Lists (ACLs)—to allow user permissions to be set on a file-by-file basis.
- DASD Limits—to allow the LAN administrator to control the use of free disk space on a per-user basis
- Separate Control Program—to allocate cache for the file system and allow cache parameters to be set independently for each volume.

Because most desktop PCs running OS/2 Warp aren't Advanced LAN Servers, and, therefore, would not stand to benefit from the additional network features included in HPFS386, there is no reason for them to run HPFS386 instead of standard HPFS.

Why is there a limit on HPFS cache size?

HPFS is an OS/2 installable file system. This IFS mechanism defines the interface between the operating system and the file system. One interface is the memory allocation available to the IFS. The memory available for use by the IFS is limited by the OS/2 kernel and is currently fixed at 2MB. To increase the HPFS cache size, either the OS/2 kernel will have to remove the limitations, or a non-IFS program will have to allocate memory for HPFS.

HPFS is based upon a 16-bit memory model (remember, all physical device drivers in the OS/2 world are 16-bit). Memory cache blocks are allocated during initialization of the IFS as OS/2 boots, based upon the cache size specified on the IFS's command line in your *config.sys* file.

HPFS uses cache memory as 2K cache blocks—thus a 2MB cache (/CACHE:2048) contains 1,024 cache blocks. Each cache block must have a corresponding BuffNode (a cache block pointer). Because HPFS is a 16-bit implementation, its global data segment (limited to 64K bytes) can only hold so many cache block pointers. For example, 1,024 cache blocks at 59 bytes per cache pointer gobbles up 59K of the 64K global data segment.

How should you set the HPFS cache parameters?

One of the most powerful features

Table 4: Guidelines for setting cache parameters for systems running OS/2 2.x or later.

HPFS Partitions Only

RAM	Cache	Lazy Writes	MaxAge	DiskIdle	BufferIdle	Other
16MB	2048	/lazy:on	40000	30000	20000	REM out the DISKCACHE statement
12MB	1536	/lazy:on	40000	30000	20000	REM out the DISKCACHE statement
8MB	1024	/lazy:on	40000	30000	20000	REM out the DISKCACHE statement

FAT Partitions Only

RAM	Disk Cache	Lazy Writes	MaxAge	DiskIdle	BufferIdle	Other
16MB	2048	/LW	N/A	N/A	N/A	REM out IFS=HPFS.IFS statement
12MB	1536	/LW	N/A	N/A	N/A	REM out IFS=HPFS.IFS statement
8MB	1024	/LW	N/A	N/A	N/A	REM out IFS=HPFS.IFS statement

HPFS & FAT Partitions with HPFS Active

RAM	Cache	Lazy Writes	MaxAge	DiskIdle	BufferIdle	Range for DiskCache Value
16MB	2048	/lazy:on	40000	30000	20000	512 - 1024
12MB	1536	/lazy:on	40000	30000	20000	256 - 512
8MB	1024	/lazy:on	40000	30000	20000	128 - 256

HPFS & FAT Partitions with FAT Active

RAM	Cache	Lazy Writes	MaxAge	DiskIdle	BufferIdle	DiskCache
16MB	1024	/lazy:on	40000	30000	20000	2048
12MB	768	/lazy:on	40000	30000	20000	1536
8MB	512	/lazy:on	40000	30000	20000	1024

available in the High Performance File System is the optional cache-tuning parameters for lazy writes—that is, postponing writing data to the hard disk for a short period of time to maximize the applications' performance. These can significantly enhance system performance.

The FAT cache only allows you to turn lazy writes on or off. The HPFS cache lets you also adjust the settings for **MaxAge**, **DiskIdle**, and **BufferIdle** on-the-fly, using the **CACHE** command followed by the appropriate parameter:

MaxAge—Elapsed time (in milliseconds) that can expire since the last write to your physical disk occurred, regardless of how many times the data held in the HPFS cache buffers may have been updated.

DiskIdle—Elapsed time (in milliseconds) that the disk controller can be inactive before the lazy writer will write data held in HPFS cache buffers to your physical disk, regardless of the **MaxAge** and **BufferIdle** settings.

BufferIdle—Elapsed time (in milliseconds) before the lazy writer writes the data held in HPFS cache buffers to your physical disk.

We recommend that the number of milliseconds specified for **MaxAge** be greater than that specified for **DiskIdle**, and, that the number of milliseconds specified for **DiskIdle** be greater than that specified for **BufferIdle**. There are no "correct" settings that are applicable for every system. They vary depending upon the type and volume of disk I/O on a system.

The guidelines in Table 4, for setting cache parameters, apply to any system running OS/2 2.*x* through OS/2 Warp. To determine which settings would be best for your system, select the appropriate section of Table 4: Make your choice based on whether your system is formatted as HPFS-only or FAT-only, or using both with either HPFS or FAT "active."

For our purposes, **active** and **passive** are descriptors of the way a partition is used. If it is seldom used, consider it **passive**. If a lot of disk-intensive I/O occurs on the partition, consider it **active**. Because swapper I/O does not pass through the normal system cache, it does not have any bearing on whether you should consider a drive **active**.

Remember, the right setting for your particular system will depend upon the type of applications you are running and the amount of RAM in your computer system. However, the higher the values are for the **MaxAge**, **DiskIdle**, and **BufferIdle** parameters, the more latitude the lazy-writer will have as it prioritizes requests.

Finally, based upon the file system and cache-tuning testing we did in conjunction with developing DCF/2, we discovered the following to be true:

- HPFS actually requires 128 to 130K of committed memory, as opposed to the widely perceived 512K. As cache size increases to 2MB, this requirement increases as well, up to a maximum of about 240K.
- The optimal cache size seems to be 1,536.
- When comparing the relative merits of HPFS versus FAT: On partitions of identical size, HPFS gives you about

Other HPFS Benefits

The High Performance File System was designed specifically to address key issues such as lost storage. But HPFS offers more than better handling of slack space. Other advanced features that make HPFS a superior file system are: integrated Extended Attributes, multithreaded I/O, and strategic allocation of directories.

Integrated Extended Attributes

Extended Attributes contain information about a file. File attributes on a FAT volume are limited to simple read-only, system, hidden, and archive flags. File attributes on an HPFS volume are called Extended Attributes (EAs) because they provide not only these basic flags but also allow for highly generalized file information (similar to environment variables) to be kept with the file.

We feel that the single most important feature of HPFS is the way it integrates a file's Extended Attributes with the file itself rather than placing them in a separate file. For example, on a FAT-formatted volume, a file's EAs are stored in an external file, EA_DATA._SF; on our 450MB FAT-formatted volume, that EA_DATA._SF file is a 5MB file—scattered all over the volume.

By contrast, on an HPFS-formatted volume, file-related information is stored directly within the file's basic disk structure. Space permitting, the file's EAs are contained directly in a file or file directory's Fnode. Otherwise, EAs are kept in normal disk sector storage in runs pointed to by the Fnode allocation data structures.

Why are integrated EAs so important? Because an advanced operating system like OS/2 uses EAs to maintain key information about certain files. In particular, many of the operating system components require specific extended attributes—on a FAT-formatted boot disk, where these EAs are contained externally in the EA_DATA._SF file, the EAs may become cross-linked. Unfortunately, the prescription for recovery usually calls for a complete reinstall of the operating system. Because extended attributes on an HPFS-formatted volume are integrated, they cannot be inadvertently cross-linked.

Multithreaded I/O

The benefit of HPFS's multithreaded I/O is improved system performance. Multiple operations happen concurrently, in the background, or when the disk controller is idle. Disk I/O is distributed evenly and the processor used more efficiently.

HPFS uses caching extensively. Caching involves allowing data being written to disk to actually be written to high "memory cache blocks," which are then later "lazy-written" to physical disk storage.

Strategic allocation of directories

FAT design is based upon a linear table of entries that must be searched from the beginning when a file or directory is requested. By contrast, HPFS places key directory and file information near the seek-center of the drive and employs a highly efficient, binary tree mechanism in search operations. The result is vastly reduced disk head movements and search times, especially on large volumes.

In addition, built into the HPFS implementation is special file and directory name caching, known as the "look-aside" naming cache. Each directory and file name in the cache is represented by a check-sum (a single 32-bit value). When the cache is searched for a particular directory or file name, the search requires only one single, 32-bit compare, as opposed to comparing every byte, byte-by-byte, of a potentially *very* long file name, in order to determine if it is target directory or file. This makes searches of HPFS volumes faster and more efficient.

15% more space, and performance is about 28% better.

- Instead of continuing to increase performance, a **DISKCACHE** value in excess of 2,048 seems to degrade performance rather than improve it.

Do any other operating systems use HPFS? What about NTFS?

Microsoft's *Windows NT* supports HPFS. There are also third-party developers who have created HPFS-compatible drivers for DOS, Linux (a new version of the UNIX operating system), and other operating systems. Users of Windows NT probably use a similarly named file system, NTFS, the New Technology File System, rather than HPFS.

HPFS obviously predates Windows NT and its NTFS. Both share a common goal: They are designed to compensate for the inadequacies of the file allocation table in light of today's multigigabyte disks and multitasking operating environments. They provide improved performance on large volumes, more efficient data storage, and greater reliability than the FAT file system.

According to Microsoft's Gary Kimura, co-architect and developer of NTFS, several significant differences exist between NTFS and HPFS. First and foremost is that NTFS is truly a 64-bit file system capable of supporting almost infinitely large files. Another major difference is that NTFS is a recoverable file system—NTFS journals disk transactions so completely that **CHKDSK** is unnecessary. On the other hand, NTFS takes after the FAT system in that it doesn't always store data at 512-byte sector granularity; NTFS uses a cluster-like storage unit, which grows in size as the size of the volume grows.

Another factor in "comparing" Windows NT's NTFS with OS/2's HPFS is that Microsoft has incorporated an extremely efficient file-by-file compression subsystem into NTFS. IBM has not elected to provide compression with HPFS.

When should and shouldn't I use HPFS?

Some people insist that you should always use HPFS. Others caution that if backward compatibility with DOS is an issue, you may find a mixture of FAT and HPFS better suited to your needs.

If you use large databases, HPFS is preferable to FAT. Databases are usually large, randomly stored files. FAT is not well-suited to perform searches on such files as efficiently as is HPFS.

Is your system likely to crash on a regular basis, perhaps because you're developing and testing systems-level software? Our own experience in developing device drivers—and the improper shutdowns that are inherently a part of device driver development—leads us to suggest that a FAT-formatted volume is preferable in such a setting. The time HPFS needs to check an improperly stopped HPFS-formatted drive offsets any perceived performance benefits. In this case, you would find that you can get back to work faster if your system is FAT formatted rather than HPFS.

As a developer, is there anything I should consider when dealing with HPFS?

Opinion on this issue seems divided, too. One HPFS developer at IBM said

no, while another suggested that there are parameters that developers should take into account: NEWNAME and FILESIZE.

The NEWNAME parameter allows OS/2 to identify a program as compatible with HPFS support for long file names. This parameter is used when linking the program. The FILESIZE parameter helps minimize fragmentation of the data stored on an HPFS-formatted volume. If the FILESIZE parameter is set (when the **DosOpen** function is used to create a file) to whatever size the developer anticipates the file will have when closed, then HPFS knows how much space to allocate for the file and does so accordingly.

Conclusion

The High Performance File System is designed to address the needs of today's OS/2 user. It is capable of managing vast amounts of data on large volumes in a multitasking environment far more efficiently than FAT, in part due to:

- Storage allocation, based upon 512-byte sector boundaries, regardless of volume size that minimizes the portion of your disk lost to slack space.
- Multithreaded design, extensive use of sophisticated caching techniques, and the lazy writer, which allow the file system to provide significantly enhanced system performance.
- Strategic allocation directories and files that allow HPFS to perform complex search operations with minimal disk head movement and greater efficiency than FAT.
- Design and data structures that allow it to allocate and preallocate contiguous file space, minimizing the disk's tendency to become fragmented.
- Support of long file names of up to 255 characters, which provides far greater file- and directory-naming flexibility than does the traditional eight-dot-three DOS convention.
- Integrated extended attributes for both the operating system and applications within its own data structures that increase the system's robustness and reliability.
- Volume integrity and data reliability that are managed while a volume is in use.

While DOS and its FAT file system have served users well, neither was ever intended for use on today's super-fast PCs with multigigabyte hard disks. Operating systems of the 1990s, like OS/2 Warp, demand a file system that can offer the features HPFS provides today.

Acknowledgements

We would like to extend our thanks to both Felix Miró and Doug Azzarito of IBM and to Gary Kimura of Microsoft for taking time to review some of the questions posed here and to offer suggestions and technical expertise. OS/2

Max and Sandy Eidswick are cofounders of Proportional Software. Max is a system architect, experienced with the DEC VAX systems and their multitasking operating environment. He designed and wrote Proportional's DCF/2, and Sandy wrote the online documentation and user's manual. Max is now providing consulting services for low-level device driver development. You can reach them at 71333.2765@compuserve.com.